ARPA ORDER NO.: 189-1

R-563-ARPA

August 1971

The ISPL Language Specifications

R. M. Balzer

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

46

MISSING PAGE NUMBERS ARE BLANK AND WERE NOT FILMED

DOCUMENT CONTROL DATA

1. ORIGINATING ACTIVITY The Rand Corporation	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

THE ISPL LANGUAGE SPECIFICATIONS

4. AUTHOR(S) (Last name, first name, initial)

Balzer, R. M.

5. REPORT DATE	6a. TOTAL NO. OF PAGES	6b. NO. OF REFS.
August 1971	49	—

7. CONTRACT OR GRANT NO.	8. ORIGINATOR'S REPORT NO.
DAHC15 67 C 0141	R-563-ARPA

9a. AVAILABILITY/LIMITATION NOTICES	9b. SPONSORING AGENCY
DDC-A	Advanced Research Projects Agency

10. ABSTRACT

The syntax and semantics of the Incremental System Programming Language, designed for use on its own computer, the ISPL machine (described in R-562). Together the language and the machine provide a complete programming laboratory environment. The syntax used to describe ISPL is APAREL (described in RM-5611), which is similar to BNF but allows imbedded alternatives. ISPL is incrementally compiled, resembles PL/I, and allows hierarchical systems to be built by providing capabilities for scheduling core and central processing unit resources, interrupt handling, and interprocess communication. Ports, the new interprocess communication facility (described in R-605), enables communication between a program and the files, terminals, physical devices, and monitor programs. Extensive debugging facilities include dynamic record verification of all pointers. The language specifically includes the facilities needed by the control program, and the machine provides many of the facilities normally implemented in software. The file system is described in R-603.

11. KEY WORDS

Computer Programming Languages
File Structure and Management
ISPL

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand or of ARPA.

ARPA ORDER NO.: 189-1

R-563-ARPA

August 1971

The ISPL Language Specifications

R. M. Balzer

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

PREFACE

This report describes the Incremental System Programming Language (ISPL), which was designed, together with the ISPL machine, to provide a programming laboratory at Rand. ISPL is an incrementally compiled system programming language containing facilities for scheduling, resource allocation, and interrupt handling. The report should be read with its companion paper, R-562-ARPA, *The ISPL Machine: Principles of Operation*, for a clear picture of the ISPL system. However, both reports should be treated as specification documents only, as the system has not yet been implemented.

Work on ISPL was sponsored by the Department of Defense's Advanced Research Projects Agency (ARPA) as an integral part of both Rand's and the client's overall program to explore current computer technology. The present report should be of interest to those concerned with computer languages and system design.

SUMMARY

The design of the ISPL language has been integrated with the design of the machine on which it runs. Together, the machine and the language comprise a complete system for producing a programming laboratory. Facilities incorporated into the language allow hierarchical systems to be built by providing capabilities for scheduling core and central processing unit resources, handling interrupts, and interprocess communication. Ports, the interprocess communication facility, enable communication between a program and files, terminals, physical devices, and monitor programs. The language is incrementally compiled and includes extensive debugging capabilities, such as dynamic record verification of all pointers.

ACKNOWLEDGMENTS

Many of the ideas contained herein arose during the ISPL study group meetings. As such, it is impossible to individually credit each idea, but my thanks to the members: Richard Bisbey, Rod Fredrickson, Bill Josephs, Larry Lewis, and Tom Wall.

My special thanks to Bill, who started the project with me and helped crystallize many of the notions upon which the ISPL machine and language are based.

CONTENTS

I. INTRODUCTION

This report describes the syntax and semantics of the Incremental System Programming Language (ISPL). The language was designed with the machine on which it runs. The ISPL machine is defined in a companion report, R-562-ARPA, *The ISPL Machine: Principles of Operation* [1]. Both reports should be read together for a clear picture of the total ISPL system, which was designed to provide a programming laboratory at Rand. However, both reports should be treated as specification documents only, as the system has not yet been implemented.

The syntax used to describe ISPL is APAREL [2], which is similar to BNF [3] except that imbedded alternatives are allowed (separated by '|' signs) and the ARBNO function represents one or more occurrences of its first argument separated by its second argument.

The report is divided into two main sections. The first describes all the data types and the legal operations upon them. The second presents each statement in the language and describes its semantics.

II. ISPL SYNTAX

IDENTIFIERS

Identifiers are composed of upper- and/or lower-case letters, digits, and the three special characters '_' (underscore), '#' (number sign), and '@' (at symbol). Identifiers must start with a letter or special character; they may not contain any imbedded blanks; and they must be between 1 and 32 characters long.

Format:

[identifier: ¬digit arbno(letter|special_character|digit,'')]
[letter: A|B|C...Y|Z|a|b|c...y|z]
[special_character: _|#|@]
[digit: 0|1|2...8|9]

EXPRESSIONS

Expressions, as defined in PL/I, are scalar expressions [4]. They are composed of identifiers and operators. During expression evaluation, argument-passing, and assignment, *only* the following conversions are *implicitly* done by ISPL:

Integer (4) ↔ Integer (2) ↔ Integer (1)
Character ↔ character varying

[operators: '+'|'-'|'*'|'/'|'↑'|'←'|'→'|'||'|'&'|'|'|'θ'|'¬'|
'='|'≠'|'<'|'>'|'≤'|'≥'|'¬='|'¬>'|'¬<'|mod]

LABELS

Two kinds of names can be attached to a statement: statement names and statement labels. A statement name is used to match levels (the beginning and ending of compound statements). If a statement name occurs with a level-ending statement, all unended levels up to and including the named level are ended. Statement labels are used as an operand in a RESUME statement (from within an ON-UNIT). Statement names (except those used to begin a procedure level) may be reused as often as desired. Statement labels must be unique within a compilation (defined below).

Format:

[statement_name: identifier ':']
[statement_label: '('identifier'):']

STATEMENTS

Statements are sequences of keywords and expressions. There are three kinds of statements: (1) simple statements, (2) level-beginning statements, and (3) level-ending statements. Although statements are independent of line- or card-image boundaries, statements extending across such boundaries must have a CNTL character at the end of every line that is not the end of the statement. This is done so that the text editor knows where statements end without having to do a complete parse of the input. Statements may end with a semicolon. After a semicolon, all text on a line is a comment.

Format:

[Statement:

```
< <statement_label | > simple_statement
| <statement_name | > <level_beginning_statement
                      |level_ending_statement> > < ';' <comment | > | > ]
```

DATA ORGANIZATION AND STORAGE CLASSES

ISPL provides two storage classes: STATIC, which is allocated at compile time for the entire life of the program; and BASED, which is explicitly CREATED and DESTROYED by the user. The elements of STATIC storage are either the primitive data-types of ISPL (i.e., INTEGER, CHARACTER string, SEMAPHORES, PORTS, etc.), or arrays of these data types. However, the elements of BASED storage are aggregates of the primitive data-types called RECORDS. A record's components may be any of the primitive data-types, or arrays of these data types. They are individually named and need not all be of the same primitive data-type.

The declared name of the record is called its RECORD_TYPE and is used to identify explicitly created instances of this type of record. Since the user may explicitly create several instances of a RECORD_TYPE, a method exists to access each instance.

A POINTER is an ISPL primitive data-type that references specific instances of RECORD_TYPES. Each time an instance of a RECORD_TYPE is created, a pointer is set to reference it. The syntax for this pointer-referencing is:

pointer_variable → record_component

The pointer references a specific instance of a record_type and the named component within that instance is accessed. The pointer referencing a desired instance may itself be a component of an instance of a record_type and so must be pointer-referenced:

$\text{pointer\_variable}_1 \rightarrow \text{pointer\_variable}_2 \rightarrow \text{record\_component}$

Such pointer-referencing can be extended to any number of levels. One instance of each RECORD_TYPE is called CURRENT and is the instance referenced if explicit pointer referencing is not used. In syntactic descriptions throughout this report, the data name will have '_specification' appended to it whenever a data access can be either pointer-referenced or not. Thus, a PORT that might be pointer_referenced appears in syntactic descriptions as:

port_specification

DECLARATIONS

All data items must be declared before being used, either explicitly in a declaration statement or implicitly by their contextual usage.

Format:

[declare_statement: DECLARE arbno (item_declaration,',')]
[item_declaration: record_name <array_bounds RECORD|PARAMETER>|
variable_name array_bounds primitive_date_type
scope <INITIAL'('initial_value')'|>

[array_bounds: '('arbno(number<':'number|>,',')')'|]
[Scope: EXTERNAL|GLOBAL|INTERNAL|]

Each of the primitive_data_types is described in Sec. III.

Within a declare statement, all the item_declarations that are primitive_data_types and that occur after either a RECORD or a PARAMETER list are components of that RECORD or PARAMETER list and may not have a scope specified. All other item_declarations for primitive_data_types are elements of STATIC storage and may have a scope specified. GLOBAL defines a variable that is referenced by the same name as an EXTERNAL variable in some other compilation. All such EXTERNAL references are to the same, single, GLOBAL definition. INTERNAL means it is neither GLOBAL nor EXTERNAL and is the default if scope is not explicitly specified.

If an initial value is specified, when the variable is allocated, it is assigned the specified value. If no initial value is specified, the value UNINITIALIZED is assigned upon allocation; if this value is used within a program, an UNINITIALIZED_DATA program-error occurs.

III. RECORDS AND PRIMITIVE DATA-TYPES

Each of the data objects in ISPL is described below. The description includes the declaration syntax and semantics, the allowed operations on the data type, and any associated pseudo-variables or built-in functions. Pseudo-variable descriptions begin with "specifies," and built-in-function descriptions begin with "returns."

RECORDS

Declaration Syntax and Semantics

Syntax:

variable<RECORD|PARAMETER>','subelement_list

where subelement_list is a sequence of any of the other declarations syntaxs given in this section. This sequence is ended either by the appearance of another record specification or the end of the declaration statement.

Semantics:

The variable name is declared to be a record and all the declarations in the subelement_list are the elements that compose the record. The amount of storage required for a record is the sum total of all the storage required for each record element, plus whatever extra storage is required for the proper alignments. If another declaration occurs for the same record, it is assumed that the subelement_list is appended to the end of the already specified data.

The different types of records have the following interpretation:

- o Record: Allocated and freed only through AUTO, CREATE, and DESTROY commands. Current instance of record is maintained by ISPL.
- o Parameter: Used to specify the formal parameters to a procedure. It is never allocated or freed, but current instance is maintained and updated by ISPL-procedure entry and exit routines. Since arguments are passed by reference, each use of a parameter

causes an indirect access through the current parameter list. Using the parameter-list name, the parameters in a parameter list can also be accessed as a one-dimensional array. By use of the built-in array function, HIGH_BOUND, the number of parameters passed can be tested dynamically. Because each of the parameters is actually a descriptor for the passed arguments, any of the descriptor (see p. 15) built-in functions and pseudo-variables, such as TYPE, can be used.

Operations

Records can be compared for equality and assigned a value, which can be accessed. In each case, the operations are performed element by element for the entire length of the record; no conversions are performed.

Pseudo-Variables and Built-In Functions

length(record-name)--Returns the length of the record in bytes.

INTEGER

Declaration Syntax and Semantics

Syntax:

variable INTEGER < '(' < 1 | 2 | 4 > ')' | >

Semantics:

Integers can be one, two, or four bytes long. If no length is specified, two bytes are assumed. One-byte integers are always positive and consist only of a magnitude. They can be aligned on any byte. Two- and four-byte integers have a sign and magnitude and are, respectively, halfword and fullword aligned.

Operations

The arithmetic operators of '+', '-', '*', '/', and arithmetic comparisons are legal and have their normal meaning. The value of integer variables can be accessed and assigned.

Pseudo-Variables and Built-In Functions

None.

CHARACTER

Declaration Syntax and Semantics

Syntax:

variable CHARACTER '(' number ')' <VARYING |>

Semantics:

Character strings can be either fixed or varying. Both types always occupy the same (during an allocation) fixed amount of storage. Varying strings occupy a varying portion of the same, fixed, maximum amount of storage.

Operations

Concatenation ('||') can be used to add one string to the end of another. The operators or('|'), and('&'), and exclusive-or('θ') can be used to do the bit-by-bit operation on the longer length of the two strings (the shorter string is extended with zero bits).

String comparison also uses the longest length of the two strings (the shorter is extended with blanks). The individual characters are compared on the basis of the collating sequence of the machine.

String assignment to variable-length strings will set the length of the variable string to that of the assigned string value.

If the length of the new value is too large for either a fixed-length string or a variable-length string, it is assigned left-justified and the excess is truncated.

Assignment of a shorter string to a fixed-length string left-justifies the value and pads it on the right with blanks.

Pseudo-Variables and Built-In Functions

Substr(string,I,J)--Specifies a descriptor for the I*th* through (I+j-1)*th* characters of the named string.

Index(string1,string2,I,not_found_expression)--Returns the position of the first instance of string2 within string1 starting at position I of string1. If string2 does not occur at or after position I, then the not_found_expression is evaluated and returned as the value of the function. If not specified, its default value is zero. This expression may be a statement that transfers control (out or end of statement) to a higher level (before a value is returned).

length(character_string_specification)--Returns the current length in bytes of the character string.

POINTER

Declaration Syntax and Semantics

Syntax:

variable POINTER

Semantics:

The named variable is declared to be of type pointer. It is fullword aligned and occupies a fullword. The value of a pointer references an address and is composed of four parts: a segment number, an offset within that segment, a read/write capability, and a record_type. To calculate the address referenced, the segment number is used as an index to a segment table associated with the process being executed. The entry in the segment table specifies the base address of the segment and its length. If the offset specified is larger than this length, the reference is illegal. The offset is added to the base to complete the address calculation. The segment-table entry may indicate that the specified segment is not presently in core, in which case ISPL suspends the process until the desired segment is available in core.

Each process has its own segment table. It and all its descendent tasks that are not themselves processes share the same segment table.

The read/write capability is a discrete value from the read_write_capability range consisting of the values read_only, read, read_write, and modal. These values are given in decreasing order of restrictiveness; therefore, in following a pointer chain or path, the resulting read/write capability is the more restrictive of the accessing capability and the accessed capability, as for example, in the pointer chain

P1(modal)->P2(read_write)->P3(read_only)->
P4(read_write)->J

In this example, the values in parentheses indicate the read/write capability of the pointers. Some implicit pointer is used to access P1, and we assume its capability is modal. P2 is also accessed with modal capability, P3 with the more restrictive read_write capability, P4 with the most restrictive read_only, and J with this same read_only capability even though P4 has read_write capability. Thus, protection can be assured by starting with, or encountering, the proper capabilities in a pointer chain or path.

There is one exception to these precedence relationships: when a read capability encounters a read_write capability, it becomes a read_ write capability Thus, local read(only) protection can be given that, via an appropriate pointer, leads to a read_write capability. This is important for system blocks that must be protected but that lead to writable blocks in a user's space.

To allow processes to restrict the read/write capability, the RESTRICT function (see p. 12) returns a pointer with the specified, more restrictive read/write capability and with all other components the same.

To allow for finer protection, new segments can be created overlaying existing ones and a restricted capability pointer created (via RESTRICT) for that segment. All further references through the pointer, or through pointers created from that pointer, can only be to data within the newly created segment. These will have a read/write capability at least as restrictive as the original pointer.

The record_type field of a pointer contains an indication of the prototype name of which the record pointed to is an instance. This is *not* an indication of the data type of the element being referenced but of the record_type of the record being pointed to. Each declared record is assigned a unique value from the range "record_type"; this value is used to distinguish the record_type of the record instance being pointed to. The record_type component is used at run time to dynamically check the validity of the pointer reference. Whenever a pointer is used to access a piece of data, the record_type component of the pointer is compared with the record_type of the object specified in the source statement (if the object is a member of a record, the record_type of this record is used). If the record_types do not agree, a pointer_reference error occurs. If the record_type of a pointer is "undefined," no record_type checking occurs. However, use of such a pointer sets its record_type to the record_type of the object specified in the source statement.

Operations

Pointers can be compared against each other for equality and inequality (this only compares the address portion of the pointers). Their values can be accessed and assigned. Arithmetic operations can be performed on their offsets through the OFFSET pseudo-variable. Such arithmetic operations affect only the offset portion of the pointer value and also set the record_type component of the pointer to "undefined." Although arithmetic manipulation of pointer offset is allowed, it should be strongly discouraged since it disables ISPL's built-in pointer debugging capabilities. (Note that the protection mechanisms in the system are unaffected by such offset manipulations.) A conscious attempt has been made to make such offset manipulations unnecessary by including the following capabilities:

- Next and previous operations on arrays (moving through contiguously stored tables).
- Substr (manipulation part of a string).
- Move_bits (machine-representation defined type-conversions).

Pseudo-Variables and Built-In Functions

NULL--Returns a pointer to an invalid address. An attempt to reference the object pointed to by a pointer with NULL as its value causes a NULL_ACCESS program error.

SEGMENT_NUMBER (pointer)--Returns as an integer the segment_number portion of the specified pointer.

SEGMENT_LENGTH (pointer)--Returns as an integer the length of the segment referenced in the specified pointer.

OFFSET (pointer)--Specifies as an integer the offset component of the specified pointer.

CAPABILITY (pointer)--Returns the read_write_capability value for the specified pointer.

RECORD_TYPE (pointer)--Specifies the record_type of the specified pointer.

NEW_SEGMENT (pointer, length)--Returns a pointer to the newly created segment. The segment starts at the address referenced by the specified pointer and extends the specified amount. If the length specified is large enough to extend out of the segment specified by the pointer, a new segment will not be created and a null pointer will be returned. The read/write capability of the returned pointer will be the same as the read/write capability of the specified pointer. If no pointer or a null pointer is specified, the new segment does not overlay any existing segment but is a separate entity.

RESTRICT (pointer, capability)--Returns a pointer with the same components as the specified pointer except for the read/write capability, which is the more restrictive of the pointer capability and the specified capability. The specified capability must be a value from the read_write_capability range.

DISCRETE-VALUED VARIABLE

Declaration Syntax and Semantics

Syntax:

variable range_name

Semantics:

Discrete variables occupy one byte and are byte-aligned. They can take on any of the symbolic values declared to be in the named range.

Operations

Discrete variables can only be assigned or compared with other discrete variables declared to have the same range. Greater than and less than apply to the order in which the values of the range were specified. In a do-case statement, the ordinal position of the value of the discrete variable within its range is used as the value of the selector function.

Pseudo-Variables and Built-In Functions

<next|previous>(variable,end_of_range_exp)--Returns the <next|previous> value from the range of the discrete-valued variable. If the <next|previous> value does not exist, the end_of_range expression is returned as the value of the function. If not specified, its value is UNDEFINED. The expression may be a statement (OUT or ENDOF) that transfers control to some higher level.

RANGE

Declaration Syntax and Semantics

Syntax:

variable RANGE '(' range_value_list')'

where range_value_list is an arbitrary list of values (separated by commas) declared to be in the named range.

Semantics:

Each range is considered to be a unique data type and other variables can be declared to be of this type.

The values in a range are ordered by their declared position. This ordering can be used in discrete-variable comparison and do-case statements.

Every range has UNDEFINED as its lowest value. Discrete variables can be set and tested for this value. If a do-case statement is executed with an UNDEFINED value, the last (out_of_bounds) statement-group is selected.

As with records, new elements can be added to the end of a range at any time through a declaration statement specifying the range name and the new elements.

Operations

None.

Pseudo-Variables and Built-In Functions

<next|previous> (discrete_variable,end_of_range_exp)--Returns the <next|previous> value in the range of the discrete variable from the current value of the discrete variable. If the <next|previous> value does not exist, the end_of_range_expression is evaluated and returned as the value of the function. If not specified, its default value is UNDEFINED. The end_of_range_expression may be a statement that transfers control (OUT or ENDOF statement) to some higher level (before a value is returned).

DESCRIPTORS

Declaration Syntax and Semantics

Syntax:

```
variable DESCRIPTOR < '(' < type | >
  < ',' < length | > < ',' data_address | > | > ')' | >
```

or

```
variable ARRAY DESCRIPTOR '(' number ')'
```

Semantics:

The value of the variable will be a descriptor for some data value. The descriptor consists of the type, length, and data address of the value being described.

For array descriptors, the number specifies the number of dimensions in the array described.

If the type, length, and/or data address are specified, the descriptor can only be used for data of the declared attributes. In these cases, ISPL can generate much more efficient code for the use of the descriptor, especially if the type is specified. Such type-specified descriptors become a type of indirect reference.

Operations

The values described by the descriptor can be accessed and assigned through the descriptor and can involve any of the operators, pseudo-variables, and built-in functions appropriate for that type of data.

Elements of the array described by an array descriptor are accessed by preceding the desired element by:

array_descriptor (subscript_list) →

Whenever the value of the descriptor itself is desired, rather than the data it describes, the descriptor_value pseudo-variable must be used.

Descriptor values can only be (1) assigned to other descriptors, (2) passed as arguments, and (3) compared to other descriptor values for equality and inequality.

Pseudo-Variables and Built-In Functions

Descriptor_value (descriptor)--Specifies the value of the descriptor rather than the data the descriptor describes. This pseudo-variable is used whenever the descriptor_value itself is to be manipulated.

Type (descriptor)--Specifies the data type being described. It is a discrete value in the record_type range that is built up by ISPL from the declared data types. The standard data types, such as integer, character, varying, pointer, etc., are included in this range, as are such user-defined types as declared ranges and records.

Length (descriptor)--Specifies the length as an integer of the data item being described.

Data_address (descriptor)--Specifies the address as a pointer of the data item being described.

Descriptor (pointer, current_length, max_length)--If maximum length is unspecified, it is set to current length. This function returns a descriptor composed of the arguments of the function.

Array_descriptor (pointer, length, low_bound; high_bound 1, low_bound 2, high_bound 2, ..., low bound M, high bound M)--Returns an array descriptor composed of the arguments of the function.

LIST STRUCTURES

Declaration Syntax and Semantics

Syntax:

```
Variable  / primitive_data_type \  / STACK \
          \ record_name         /  < QUEUE  > '('domain_specification')'
                                   \ RING  /

  / WITH  / number                               \  STATIONS \
  \   |   \ '('arbno(station_name,',')')'        /           /
```

Semantics:

The variable is declared as the indicated type of list structure. The type of all the elements of a list structure is the same and is the primitive data-type or record_name specified. STACKS are list structures in which elements are added (PUSHED) and removed (PULLED) from the same position (STATION), called the TOP. QUEUES are list structures in which elements are added at one STATION, called the BOTTOM, and removed from another STATION, called the TOP. RINGS are list structures in which elements can be added or removed BEFORE or AFTER any STATION in the RING. The elements of STACKS and QUEUES are linked from the TOP toward the other end, whereas the elements of RINGS are linked in a circle in both the FORWARD and BACKWARD directions. All the elements of a list structure are obtained and returned to a pool of elements, called a DOMAIN. The DOMAIN used for each list structure is specified in its declaration. In addition to the fixed named stations, list structures may have as many additional movable (via STEP command) stations as desired. These are specified either by number (and accessed as an array) or by name. If not specified, STACKS and QUEUES have one movable station and RINGS have two.

Operations

STACKS, QUEUES, and RINGS can only be operated on by the PUSH and PULL operators. Their movable stations can be operated on by the STEP operator and can be assigned to other stations within the same list structure. Both fixed and movable stations can be compared with each other for equality and inequality.

The stations within a list structure can also be used to access the element or components of the element referenced by that station, i.e., a station can be used as a "pointer." The syntax is

```
                              / '(' /number      \ ')' \
list_structure_specification <       \station_name/      >  →
                              \  |                      /

              element_or_component_name
```

If a particular station is not specified, the first movable station is used.

Exa. ples: Given the declarations:

DECLARE S1 INTEGER STACK (Domain_1) WITH (movable_1,
movable_2) STATIONS,
Q1 R1 QUEUE (Domain_1) WITH 2 STATIONS,
R1 RECORD, P POINTER, I INTEGER (4);

Then the following examples are legal:

S1(movable_2) → INTEGER
Q1 (1) → I same as Q1 → I
Q1 (BOTTOM) → P → I

Pseudo-Variables and Built-In Functions

None.

STATIONS

Declaration Syntax and Semantics

Syntax:

variable <RING |> STATION

Semantics:

The declared station is a movable station. Ring stations can only be used to reference elements of a RING structure, and nonring stations can only be used to reference elements of STACK or QUEUE structures. However, STACK or QUEUE structures do not necessarily have to reference elements from a particular list structure or even from the same DOMAIN.

Operations

Same as movable stations described in list structure, above.

Pseudo-Variables and Built-In Functions

None.

SEMAPHORE

Declaration Syntax and Semantics

Syntax:

$$\text{variable} \left\langle \begin{matrix}\text{SYNCHRONOUS}\\ | \end{matrix} \right\rangle \text{SEMAPHORE}$$

$$\left\langle \text{'('assignable\_item} \left\langle \left\langle \begin{matrix}\text{STACK}\\ \text{QUEUE}\\ | \end{matrix} \right\rangle \text{'('domain\_specification')'} \right\rangle \text{')'} \right\rangle$$

where the STACK and QUEUE declarations are defined as above, and assignable_item is either any primitive data-type to which the assignment operator can be applied or a RECORD of such items.

Semantics:

Semaphores are the basic mechanism for synchronizing processes, tasks, and exclusive-execution blocks (EEBs). The V operation makes the semaphore available and the P operation makes it unavailable. If it is already unavailable, the P operation causes the issuer to wait until the semaphore has been made available, and then makes it unavailable. Scheduling, swapping, Ports, and interrupts are all based on semaphores.[†]

The variable is declared to be a semaphore. If it is a SYNCHRONOUS semaphore, then, when an unsuccessful P operation is done on it, the running EEB is considered to have exited and any lower-priority dispatchable EEBs in the running task are dispatched. If the semaphore is not SYNCHRONOUS, then an unsuccessful P is not treated as an exit and lower-priority dispatchable EEBs are not dispatched until such an exit occurs [1].

The semaphore can have data associated with it so that a successful P also returns a piece of data, such as a track number, a completion code, or a pointer to a parameter_list (see p. 20). A V operation on such a semaphore must supply the data to be returned on a P. This data must be stored; three methods are available: STACKS and QUEUES, to provide, respectively, last-in first-out (LIFO) and first-in first-out (FIFO) buffering, and unbuffered, where only one data item can be held without a data overflow.

Operations

Only the P and V operations can be performed on semaphores. The data format of these operations must be used with data semaphores.

Pseudo-Variables and Built-In Functions

None.

[†]See Ref. 1, Sec. V for a fuller explanation.

PORTS

Declaration Syntax and Semantics

Syntax:

$$\text{variable} \left\langle \begin{array}{c}\text{SYNCHRONOUS}\\ | \end{array}\right\rangle \text{PORT}$$

$$\left\langle \begin{array}{l} \text{'('} \left\langle\begin{array}{c}\text{STACK}\\ \text{QUEUE}\end{array}\right\rangle \text{'('domain\_specification'))'} \\ | \end{array}\right\rangle$$

where STACK and QUEUE declarations are defined as above.

Semantics:

Ports are a program's method of communicating with the outside world--files, terminals, and Ports in other programs. Any number of arguments can be passed or received through a Port. Although Ports are a primitive data-type, they are composed of a pointer (which references the other Port of the interconnected pair) and a data semaphore, where the data is a pointer to the argument_list passed across the Port. The data semaphore can be SYNCHRONOUS or not, and can be buffered (either stacked or queued) or unbuffered.

Operations

Ports can be CONNECTED with files, terminals, or other Ports--and can be DISCONNECTED from them. Arguments can be SENT and RECEIVED over a Port and information can be REQUESTED.

Pseudo-Variables and Built-In Functions

None.

DYNAMIC STORAGE MANAGEMENT

Declaration Syntax and Semantics

Syntax:

variable $\left\langle \begin{matrix} \text{AREA} \\ \text{DOMAIN} \end{matrix} \right\rangle$ '('number')'

Semantics:

An AREA is a contiguous block of free storage from which RECORDS are CREATED and to which they are returned when DESTROYED. A DOMAIN is a contiguous block of free storage from which elements for list structures are taken for PUSHING and to which they are returned when PULLED. A DOMAIN is divided into equal-size blocks of free space, which are used for the elements of the list structures obtained from the DOMAIN. This size is the maximum size of any element obtained from the DOMAIN.

For both AREAS and DOMAINS, the number specified is the length of the AREA or DOMAIN, in bytes.

Operations

Records can be CREATED from and returned to (DESTROYED) an AREA. List-structure elements can be PULLED from or PUSHED to a DOMAIN. Both AREAS and DOMAINS can be INITIALIZED, which makes all space within them available.

Pseudo-Variables and Built-In Functions

None.

IV. STATEMENTS

PROCEDURE

Syntax:

PROCEDURE ⟨'('Parameter_list_name')'⟩ ⟨','function_attribute⟩

Semantics:

The PROCEDURE statement indicates that all statements within the level started by the PROCEDURE statement are to be treated as a subroutine, invokable only through a CALL statement (or a function reference); this subroutine returns to the point of invocation upon termination. The PROCEDURE statement must have a unique (within the compilation) STATEMENT_NAME attached to it. It is invoked by specifying this STATEMENT_NAME.

The first of the two optional specifications in the PROCEDURE statement is the parameter specification, which specifies the name of the formal parameter list to be used for the procedure. The second option, if present, specifies the attributes of the value returned as the result of the function. The function attributes have the same format as declaration statements, except that storage class, scope, and initial attributes are not specified. The value returned may be any ISPL elementary data-type except semaphore, Port, area, or domain.

Upon entry to a procedure, the context is preserved and upon exit it is restored. This context consists of:

- o Entry point of procedure
- o Return point from procedure
- o Reference to current records upon entry to the procedure.

DO

Syntax:

DO ⟨CASE expression | CONTINUOUSLY | iterative_specification⟩

Semantics:

The DO statement specifies iteration or selection of the statements within the level started by the DO.

There are three types of DO statements:

1. DO CASE: The value of the expression is used to select one statement group from the sequence of statement groups that lexicographically follow the DO CASE statement. This group is executed and, upon completion, control passes to the end of the entire DO CASE level. A DO CASE level can only be ended by an END statement, which specifies the label associated with the DO CASE statement or the label of a level in which the DO CASE statement is contained. Therefore, for consistency, DO CASE statements must be labeled.

   Each statement group is explicitly ended by an END statement and the next statement group begins with the next statement.

   The first statement group is number one. If the value of the expression is less than one or greater than the number of statement groups within the DO CASE level, the last statement group is executed.
2. DO CONTINUOUSLY: This statement is the basic iterative looping mechanism in the language and specifies (1) that the group of statements in the level begun by the DO statement are to be executed, and (2) that upon each completion of this level, the group of statements should be re-executed. Presumably, there is some mechanism within the level that will halt this iterative execution (see p. 25).
3. ITERATIVE DO: This statement has the same syntax and semantics as the PL/I iterative DO statement.

IF AND ELSE

Syntax:

IF expression THEN

and

ELSE

Semantics:

The IF statement starts a level, called the THEN level, which is explicitly ended by either an ELSE or an END statement. The ELSE statement, if present, starts the ELSE level, which is explicitly ended by an END statement. If the ELSE statement is labeled, it ends all lexicographically preceding levels that have not been ended, up to and including the labeled level, which must be a THEN level.

The expression is evaluated and, if true, the THEN level is executed and the ELSE level, if present, is skipped. If the expression is false, the THEN level is skipped and the ELSE level, if present, is executed.

<u>END</u>

Syntax:

END

Semantics:

The END statement ends one or more levels (started by PROCEDURE, DO, or IF-THEN-ELSE statements). If it is unlabeled, it ends the lexicographically closest preceding level that has not already been ended. If a label is specified, all such unended levels are ended up to and including the level with the specified label.

The action taken upon execution of an END statement depends upon the type of level it ends. These actions are

1. PROCEDURE: Return to caller from procedure and restore caller's context.
2. DO: The end of a DO CASE statement is a no-op. The end of a DO CONTINUOUSLY or an ITERATIVE DO is a loop back to the beginning of the level within the DO. In the case of an ITERATIVE DO, it also increments and tests the control variable against the limit.
3. IF: The end of an IF statement is treated as a no-op.

OUT

Syntax:

OUT <label | >

Semantics:

The OUT statement transfers control out of the current level; that is, control continues with the statement immediately following the END statement for the level being exited. An exception is a procedure statement, for which control continues with the statement following the invocation of the procedure rather than with the statement following the end of the procedure.

If a label is specified, then the OUT statement applies to that level rather than to the current one. The specified level must be a dynamic ancestor of the OUT statement.

ENDOF

Syntax:

ENDOF < label | >

Semantics:

ENDOF behaves exactly as does OUT, except that the END statement at the end of the specified level is executed. Thus, ENDOF applied to a DO CONTINUOUSLY or an ITERATIVE DO causes looping.

CALL

Syntax:

CALL statement_name <'('arbno(expression,',')')' | >

Semantics:

The CALL statement invokes the procedure specified by the statement name, passing any arguments specified. The arguments are all passed by reference (expression arguments are passed by a reference to a temporary variable containing the value of the expression).

After the named procedure has returned, execution continues with the statement following the call.

ASSIGNMENT

Syntax:

Variable_specification ← expression

Semantics:

Note the use of the left arrow (←) as the assignment operator.

The value of the expression is assigned to the variable specified. As noted in Sec. II, the only conversions implicitly done by ISPL are

integer (4) ↔ integer (2) ↔ integer (1)
character ↔ character varying

In ISPL, *only* the following items can have a value ASSIGNED to them:

- o Integer,
- o Character,
- o Character varying,
- o Pointer,
- o Discrete-valued variables,
- o Descriptors,
- o Array descriptors,
- o Stations,
- o Ring stations,
- o Records composed only of the above items.

The following items cannot have a value ASSIGNED to them:

- o Range,
- o Stack,
- o Queue,
- o Ring,
- o Semaphore,
- o Port,

- o Area,
- o Domain,
- o Records with at least one item from this list.

SYNCHRONIZATION

Syntax:

$$\left\langle \begin{matrix} \text{IF} \\ | \end{matrix} \right\rangle \left\langle \begin{matrix} \text{P} \\ \text{WAIT} \end{matrix} \right\rangle \ \text{semaphore\_specification} \ \left\langle \begin{matrix} \text{','variable\_specification} \\ | \end{matrix} \right\rangle \left\langle \begin{matrix} \text{THEN} \\ | \end{matrix} \right\rangle$$

and

$$\left\langle \begin{matrix} \text{V} \\ \text{SIGNAL} \end{matrix} \right\rangle \ \text{semaphore\_specification} \ \left\langle \begin{matrix} \text{','expression} \\ | \end{matrix} \right\rangle$$

Semantics:

The P and V operators are Dijkstra's synchronization operators, which operate uninterrupted [5]. V increments the semaphore by one and P waits until the semaphore is positive and then decrements it by one. V thus corresponds to releasing a semaphore and P corresponds to obtaining one. Enclosing a P operation by the keywords IF and THEN causes the P operation to be performed only if it will not cause a wait. For the purposes of the enclosing IF statement, performing the P operation makes the 'if expression' TRUE and causes the THEN level to be executed. Skipping the P operation makes the 'if expression' FALSE and causes the THEN level to be skipped and the ELSE level, if present, to be executed.

Data semaphores not only provide synchronization, but also attach a piece of data to the semaphore released or obtained. In the V operation, this data is supplied as the value of the specified expression; in the P operation, the data is assigned to the variable specified.

CONNECT

Syntax:

CONNECT port_type WITH port_type

where the syntax of port_type is

$$\left\langle \begin{array}{ll} \text{port\_specification} & \\ \text{TERMINAL} & \text{terminal\_id} \\ \text{FILE} & \text{file\_name} \end{array} \right\rangle$$

Semantics:

A message path is established between the specified Ports. Data can then be passed in either direction along this path. The system automatically schedules these interconnected processes on the basis of availability of the resources required by the processes (including data coming in through a path or sent out through a path having been processed).

Ports can be terminals (logical devices through the IBM 1800), files, or named Ports in a process' program.

In ISPL, the interconnection of Ports provides a very general form of co-routines. The co-routines can be multiply connected and are data-directed; that is, rather than having explicit co-routine control commands, the availability of any required data along a path is used to coordinate and synchronize the co-routines.

The Port facility is also the method by which programs can be connected with terminals and/or files. As such, it is a form of job control and can be specified either (1) externally to, or (2) within a process.

A Port is a primitive data-type composed of a pointer and a data semaphore. The CONNECT command merely sets the two pointers to reference each other.

DISCONNECT

Syntax:

DISCONNECT port_specification

Semantics:

The message path associated with the named Port is broken (i.e., the pointers in the two interconnected Ports are set to NULL). If the Port is not connected, a run-time error results.

SEND

Syntax:

$$\text{SEND} \left\langle \begin{array}{l} \text{arbno(expression,',')} \\ \text{DUMP} \end{array} \right\rangle \text{THROUGH port\_specification}$$

Semantics:

Since the sending of a message is really a form of co-routine linkage, the method of passing the message will be the same as passing arguments to a subroutine. Namely, the number and format of arguments are established as a convention between the Ports being connected; arguments are passed by reference.

The SEND operation is equivalent to a V operation on the remote Port's (the one the specified Port is connected to) data semaphore, passing the pointer to the argument list as data.

A run-time error results if a message is sent through an unconnected Port.

The DUMP option produces a complete symbolic dump of all variables in the process and its descendent processes. This includes all instances of based records, semaphores, Ports, etc. Also included in the DUMP is a symbolic display of the invocation chains and context of each of the processes.

During the DUMP, all processes are suspended.

RECEIVE

Syntax:

$$\left\langle \begin{array}{c} \text{IF} \\ | \end{array} \right\rangle \text{RECEIVE parameter\_list\_name THROUGH port\_specification} \left\langle \begin{array}{c} \text{THEN} \\ | \end{array} \right\rangle$$

Semantics:

This statement causes a sent message to be received through the named Port. If none is available, the receiver waits until one is. Enclosing a RECEIVE operation by the keywords IF and THEN causes the RECEIVE operation to be performed only if it will not cause a wait.

For the purposes of the enclosing IF statement, performing the RECEIVE operation makes the 'if expression' TRUE and causes the THEN level to be executed. Skipping the RECEIVE operation makes the 'if expression' FALSE and causes the THEN level to be skipped and the ELSE level, if present, to be executed.

Since, as explained in the send command, a message consists of an arbitrary number of arguments, the receiver of a message must have a mechanism for manipulating each message. As with subroutine calls, a formal parameter list is used to associate actual arguments with formal parameters on a positional basis.

The receive command is equivalent to a P on the specified Port's data semaphore, assigning the received pointer as the current instance of the named parameter list.

REQUEST

Syntax:

REQUEST arbno(expression,',') AS parameter_list_name THROUGH port_specification

Semantics:

The specified arguments are sent through the specified Port and the message sent back is received in the named parameter_list. It is assumed that the program on the other end of the Port uses the sent arguments to select or specify the message returned.

DYNAMIC STORAGE ALLOCATION

Syntax:

CREATE record_specification ⟨ IN area_specification | AS A SEGMENT | ⟩

and

DESTROY record_specification

Semantics:

The CREATE statement creates an instance of the record specified. In addition, if a pointer chain was used in the record specification, then the rightmost pointer in the chain is set to reference the new instance. Otherwise, the new instance is made CURRENT.

If AS A SEGMENT is specified, then the created record is placed in a new segment just large enough for it. Otherwise, the record is created within the specified AREA or within a system-defined AREA if none is specified.

In a DESTROY statement, the record specified is destroyed and the pointer used to reference it, either explicitly in the pointer chain of the record specification or the CURRENT pointer for the record, is set to NULL. If the record destroyed was created AS A SEGMENT, the segment is destroyed.

AUTO

Syntax:

AUTO arbno(record_name,',')

Semantics:

New instances of the named records are CREATED and made CURRENT. Upon exit from the level in which the AUTO statement was issued, the instances are DESTROYED and the instances that were CURRENT before the AUTO statement are again made CURRENT.

MAKE

Syntax:

$$\text{MAKE} \left\langle \begin{array}{l} \text{record\_specification CURRENT} \\ \text{process\_variable\_specification} \left\langle \begin{array}{l} \text{ACTIVE} \\ \text{INACTIVE} \end{array} \right\rangle \end{array} \right\rangle$$

Semantics:

The ACTIVE and INACTIVE options allow the monitor of a NEW process (1) to RETRIEVE all core allocated to a process and temporarily remove the process from the system, and (2) to restore it from an inactive status [1].

The CURRENT option makes the specified instance the CURRENT one for that record_type.

PUSH AND PULL

Syntax:

```
< <PULL>         list_item > <AND> <PUSH list_item>
  <POP >                     < |  > < |           >
< |                        >
```

where list_item is

```
< stack_or_queue_structure_specification                                                      >
< <BEFORE> <ring_structure_specification>  '('  <expression  >  ')'                          >
< <AFTER > <ring_station_specification  >       <station_name>                                >
```

Semantics:

Either half of this operation can be omitted (the AND connective is used only when both halves are present). If either half is omitted, the omitted half's operation is performed on the FREE_ELEMENT_STACK in the domain. The PULL or POP operation removes an element from the specified list structure. For stacks and queues, the element is specified by the TOP station. For rings, it must be explicitly specified as the element either BEFORE or AFTER a particular station. If the element is PULLED from the FREE_ELEMENT_STACK, it is initialized as specified by the element's declaration. The indicated or implied station is updated to reference the next element or link (for rings) in the list structure. If any other stations in the stack, queue, or ring reference the PULLED one, they are set to NULL. The PUSH operation adds the PULLED element to the specified list structure. For stacks, it is added BEFORE the TOP station; for queues, it is added after the BOTTOM station; and for rings, it is added, as explicitly specified, as the element either BEFORE or AFTER a particular station. In each case, the implied or explicit station is updated to reference the PUSHED element.

STEP

Syntax:

```
STEP station_item <FORWARD >
                  <BACKWARD>
                  < |      >
```

where station_item is

```
 / stack_or_queue_or_ring_structure_specification \  / '(' / expression   \ ')' \
 \ station_specification                          /  \  |  \ station_name /     /
```

Semantics:

If no station is specified, the first movable station is used. The indicated station is updated to reference the next element in the specified list structure in the indicated direction (FORWARD is implied if neither is specified). Only rings may be stepped BACKWARD. If an attempt is made to STEP beyond the end of a stack or queue, the station references NULL.

STATION ASSIGNMENT

Syntax:

station_item ← station_item

where station_item is defined above.

Semantics:

The station_item specified on the left of the assignment arrow is set equal to the station_item on the right. If the station specified on the left is part of a list structure (i.e., not an independently declared station), the station on the right must be a station in the same list structure. On the right hand side, TOP may be specified as the station_name for either STACKS or QUEUES, and BOTTOM may be specified as the station_name for QUEUES.

INITIATE

Syntax:

```
                           / '('arbno(expression,',')')' \  / ','EXCLUSIVE   \
INITIATE statement_name   <                               >< ','INDEPENDENT  >
                           \  |                          /  \ ','NEW         /

 / ','AS process_variable \  / ','PRIORITY expression \  / ','ENABLE \
 \  |                     /  \  |                     /  \  |        /
```

Semantics:

The INITIATE statement invokes the named procedure and passes parameters to it just as a call statement does. However, control also logically proceeds with the statement following the INITIATE statement, without waiting for the initiated procedure to return. Thus, INITIATE produces a control "fork" or parallel path in the process. The optional priority expression specifies the relative priority to be assigned to the initiated procedure.

The initiated procedure starts a separate control path, or fork. This fork can have three separate relationships with the one that issued the command. First, the initiated procedure can be an INDEPENDENT asynchronous fork sharing the addressing space with the initiator. Synchronization of the two forks is accomplished through P and V operations on semaphores. In conflict situations where both forks are able to run, the assigned priorities are used to establish precedence. This corresponds to what most systems call multitasking or multiprogramming and is the default if one of the other two options is not specified. This case corresponds to creating an INDEPENDENT EXECUTION BLOCK (IEB) in the ISPL machine [1], and is called a TASK.

The second relationship is established by the EXCLUSIVE option. The initiated procedure represents a separate flow of control but cannot logically be running at the same time as the initiating control flow. The two are mutually exclusive and once one starts to run the other is not given control when the first waits for an asynchronous event (semaphore). They are scheduled on a priority basis, as are independent forks, but a lower-priority exclusive unit is given control only when the higher-priority one has issued a P (wait) operation on a SYNCHRONOUS semaphore. This case corresponds to EEBs in the ISPL machine and is used to implement ON-UNITS and synchronous co-routines synchronized by semaphores and/or Ports. For an EEB to act as an ON-UNIT, it must P the desired semaphore, which causes it to be suspended until the interrupt occurs.

The third relationship is established by the NEW option. This creates a completely separate asynchronous process with its own separate address space. The process that issued the command *must* act as the monitor for the initiated process, scheduling and DISPATCHING it, allocating core to it, and handling its segment faults and monitor requests. This option is used by the MAJOR MONITOR to create separate processes for each user.†

In each case, the end of the fork created is caused when the initiated procedure returns.

TERMINATE

Syntax:

TERMINATE process_variable

Semantics:

The flow path (whether NEW, EXCLUSIVE, or INDEPENDENT) referenced by the process variable is removed from the system, as are all its descendent flow paths.

PROCESS SUSPENSION

Syntax:

$$\left\langle \begin{matrix} \text{ENABLE} \\ \text{DISABLE} \end{matrix} \right\rangle \left\langle \begin{matrix} \text{process\_variable} \\ \text{EXCLUSIVE SWITCHING} \\ \text{INDEPENDENT SWITCHING} \end{matrix} \right\rangle$$

Semantics:

If a process variable is specified, the referenced EEB or IEB, and all its descendent IEBs, are ENABLED or DISABLED. A DISABLED block cannot be executed, whether or not it is ready to run and no matter what its priority. ENABLING a block makes it eligible for execution when it is ready to run and has a high enough priority.

DISABLING EXCLUSIVE SWITCHING prevents any EEBs within the issuing IEB from interrupting the current EEB until EXCLUSIVE SWITCHING is again ENABLED. DISABLING INDEPENDENT SWITCHING prevents any IEBs

---

†See Ref. 1 for a fuller description of monitor responsibilities and capabilities.

within the issuing process from interrupting the current IEB until INDEPENDENT SWITCHING is again ENABLED.[†]

RESUME

Syntax:

RESUME AT statement_label

Semantics:

The EEB interrupted by the running EEB is resumed at the specified statement_label when it is next dispatched. The specified statement_label must be in an active level in that EEB, i.e., the *level* in which the statement_label occurs must be reachable from the point of interruption via an OUT statement. Note that this statement does not end the execution of the interrupt block, but only affects where the interrupted block will be resumed.

INITIALIZATION

Syntax:

$$\text{INITIALIZE} \left\langle \begin{array}{l} \text{domain\_specification} \\ \text{area\_specification} \\ \text{record\_specification} \end{array} \right\rangle$$

Semantics:

When variables are allocated, they are initialized either as specified in the declaration of the variable or as the UNINITIALIZED value if initialization has not been specified. The initialize statement records allow areas and domains to be reinitialized as desired.

PAUSE

Syntax:

PAUSE

[†]See Ref. 1 for a fuller explanation.

Semantics:

The process issuing the PAUSE is suspended (DISABLED) and the user at a terminal, who is the dynamic ancestor of the process, is informed of the pause. The process can be restarted via the ENABLE command.

This facility is included in ISPL to allow users to plant "breakpoints" in their programs as an aid to on-line debugging.

DISPATCH†

Syntax:

DISPATCH process_variable_specification

Semantics:

Execution of the specified NEW process, which must have been INITIATED by the issuing process, is resumed. Execution of the dispatcher is suspended until the dispatched process becomes undispatchable or an interrupt occurs for the dispatcher. DISPATCH is used by a monitor to allocate execution to its subprocesses.

REAL CORE ALLOCATION

Syntax:

GIVE segment_number TO process_variable_specification AS segment_number

and

RETRIEVE segment_number

Semantics:

The real core corresponding to the specified segment is GIVEN to the specified subprocess as the specified segment in his addressing space; it is GIVEN by the issuer (who must be the monitor of the subprocess). The

---

†DISPATCH and REAL CORE ALLOCATION are used by monitors to allocate execution and core to their subprocesses. They correspond directly to ISPL-machine primitive operations. See Ref. 1 for a fuller discussion.

contents of the subprocess' segment are automatically restored from secondary storage by the ISPL machine. The given segment is marked so that the real core can be RETRIEVED from the subprocess merely by specifying the given segment number. If the contents of the subprocess' segment have been modified since they were given, the RETRIEVE operation causes the ISPL machine to automatically save the new contents on secondary storage for use the next time the segment is given. GIVE and RETRIEVE are used by a monitor to allocate real core [illegible] its subprocesses.

REFERENCES

1. Balzer, R. M., *The ISPL Machine: Principles of Operation*, The Rand Corporation, R-562-ARPA, August 1971

2. Balzer, R. M., and D. J. Farber, "APAREL--A Parse-Request Language," *Communications of the ACM*, Vol. 12, No. 11, November 1969, pp. 624-631.

3. Backus, J. W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proc. Intl. Conf. on Information Processing*, UNESCO, 1959, pp. 125-132.

4. *IBM System/360, PL/1 Reference Manual*, IBM Corporation, Form C23-8201, Poughkeepsie, N.Y., 1968.

5. Dijkstra, Edsger W., "The Structure of the 'THE' - Multiprogramming System," *Communications of the ACM*, Vol. 11, No. 5, May 1968, pp. 341-346.